

Lecture 2: Getting Started

Agenda:

- Getting started
 - Sorting & insertion sort
- Analysis
 - Analysis using RAM
 - What Θ means informally
 - Asymptotic running time
 - Worst/average/best case
 - Insertion sort analysis

Reading:

- Textbook pages 5 – 27

Getting started

- Algorithm:

A well-defined computational procedure (namely, a sequence of elementary computational steps) which takes an input value and produces an output value, according to a special mathematical function.

- Describing algorithms: pseudocode

- Pseudocode example

```
input:  integers  $a, b$ 
output:  $a \times b$ 
```

```
 $sum \leftarrow 0$ 
for  $j \leftarrow 1$  to  $b$  do
     $sum \leftarrow sum + a$ 
return  $sum$ 
```

- Pseudocode conventions

- indentation indicates block structure
- while/for/repeat/if/then/else
- loop counters retain values !!!
- ** or ▷ comment
- variables are local (unless stated otherwise)
- parameters passed by value
- comparison: boolean “short circuit” evaluation
e.g. $j > 0$ AND $A[j] < key$

Sorting

- Input: a sequence of n numbers $(a_1, a_2, \dots, a_n)$
- Output: a permutation $(a'_1, a'_2, \dots, a'_n)$ such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$
- Insertion sort — one sorting algorithm (more coming soon)
 - Idea: repeatedly insert $A[j]$ into sorted sublist $A[1..j-1]$
 - How to insert?
 - * search for insert location in sequence, $A[j-1]$, $A[j-2]$,
 $\dots$
 - * move contents to the right during the search

- Pseudocode

```
InsertionSort(A) **sort  $A[1..n]$  in place
```

```
for  $j \leftarrow 2$  to  $n$  do
   $key \leftarrow A[j]$  **insert  $A[j]$  into sorted sublist  $A[1..j-1]$ 
   $i \leftarrow j-1$ 
  while ( $i > 0$  and  $A[i] > key$ ) do
     $A[i+1] \leftarrow A[i]$ 
     $i \leftarrow i-1$ 
   $A[i+1] \leftarrow key$ 
```

- Insertion sort trace: (53, 21, 47, 62, 14, 38)

53	<u>21</u>	47	62	14	38	** $j \leftarrow 2$
21	<u>53</u>	47	62	14	38	** end of this iteration
21	53	<u>47</u>	62	14	38	** $j \leftarrow 3$
21	47	<u>53</u>	62	14	38	
21	47	53	<u>62</u>	14	38	
21	47	53	<u>62</u>	14	38	
14	21	38	47	<u>53</u>	62	** output permutation

Analysis of insertion sort

- Running time
 - How to measure it? implement and run?
 - Problem: run time may vary (language, machine, code, environment)
 - Problem: not always feasible
 - Idea/Solution:
 - * select theoretical computer model
 - * estimate running time on model by the number of cycles

Model of computation: RAM

- RAM: random access machine (Page 21-22)
- Components
 - IT: input tape (read-only)
 - OT: output tape (write-only)
 - CU: computation unit (inc. program)
 - M: memory locations (each can store an integer)
M[0], M[1], M[2], ...
 - Program: fixed user-defined instruction sequence
- Properties
 - CU: instructions (each usually via register/accumulator):
 - move data between memory
 - compare data and branch
 - binary arithmetic operation
 - read from IT to memory
 - write from memory to OT
- Example RAM instructions for $z \leftarrow x + y$:

```
M[0] := M[@x] **wherever x is
M[1] := M[@y] **wherever y is
add      **M[0] := M[0] + M[1]
M[@z] := M[0] **wherever z is gets sum
```

Analysis of RAM programs

- Time — instructions executed
- Space — memory locations used
- Example — running time to multiply $a \times b$
- Answer *depends* on sizes of a and b
 - if a , b , $a \times b$ each fits into one RAM memory word, then constant number of RAM instructions, so constant time
 - if not
 - * need multiple words to represent a , b , $a \times b$
 - * can show number of RAM instructions proportional to (words to represent a) $\times$ (words to represent b)
 - * with k bits per word, number a needs using $\frac{\lg a}{k}$ words
 - * time proportional to $\lg a \times \lg b$ (since k constant)
 - * write $\Theta(\lg a \times \lg b)$ time
- Types of RAM models
 - log cost RAM
 - * assume numbers may not fit into one memory word
 - * $a \times b$ takes $\Theta(\lg a \times \lg b)$ time and $\Theta(\lg a + \lg b)$ space
 - uniform cost RAM
 - * assume each number fits into one memory word
 - * $a \times b$ takes $\Theta(1)$ time and space
 - Unless otherwise stated, assume uniform cost RAM

Analysis of insertion sort

- Running time
 - Model of computation: RAM
 - Problem: run time varies with input
 - Idea/Solution:
 - * estimate worst/average/best case performance
 - * make estimate a *function* of input *size*
 - * sorting: input size usually takes as the number of keys
 - * guarantee of performance

Kinds of analysis

- Worst case
 - $T(n)$ — maximum time over all inputs of size n
- Average case
 - Must specify input distribution over which average computed
 - Most common: assume uniform (a.k.a. equiprobable) input distribution (all inputs of size n equally likely)
 - Useful but usually difficult
- Best case
 - Useful for lower bound
 - Not otherwise useful: any algorithm can be modified to have fast best case (by adding: if input is particular case then return particular answer)

Lecture 2: Getting Started

Analysis of insertion sort

InsertionSort(<i>A</i>)	<i>cost</i>	<i>times</i>
for $j \leftarrow 2$ to n do	c_1	n
$key \leftarrow A[j]$	c_2	$n - 1$
$i \leftarrow j - 1$	c_3	$n - 1$
while ($i > 0$ and $A[i] > key$) do	c_4	$\sum_{j=2}^n t_j$
$A[i + 1] \leftarrow A[i]$	c_5	$\sum_{j=2}^n (t_j - 1)$
$i \leftarrow i - 1$	c_6	$\sum_{j=2}^n (t_j - 1)$
$A[i + 1] \leftarrow key$	c_7	$n - 1$

t_j — number of times the while loop test is executed for j .

$$T(n) = c_1 n + (c_2 + c_3 - c_5 - c_6 + c_7)(n - 1) + (c_4 + c_5 + c_6) \sum_{j=2}^n t_j$$

Running time

- Best case: list is already sorted ($t_j = 1$)

$$T(n) = a \times n + b$$

- Worst case: list is reverse sorted ($t_j = j$)

$$T(n) = a \times n^2 + b \times n + c$$

Why worst case analysis?

- upper bound — guarantee
- occurs fairly often
- average case roughly as bad as worst case

Lecture 2: Getting Started

Have you understood the lecture contents?

well	ok	not-at-all	topic
☐	☐	☐	diff between problem & instance
☐	☐	☐	insertion sort algorithm
☐	☐	☐	pseudocode convention
☐	☐	☐	alg analysis in general
☐	☐	☐	RAM (uniform cost, log cost)
☐	☐	☐	insertion sort analysis